LOGIC PROGRAMMING AND DIGITAL CIRCUIT ANALYSIS

W. F. CLOCKSIN

▷ We show how the unique character of logic programming can be exploited for the purpose of specifying and automatically reasoning about electrical circuits. Although propositional logic has long been used for describing the truth functions of combinational circuits, the more powerful predicate calculus on which logic programming is based has seen relatively little use in design automation. Previous researchers have introduced a number of techniques similar to logic programming, but many of the useful consequences of the methodology have not been demonstrated. We describe particular consequences of using this method for writing directly executable specifications of circuits, including the use of quantified variables, verification of hypothetical states, and sequential simulation. We have used these methods to solve problems in gate assignment, specialization of standard definitions, and determination of signal flow. ◁

1. INTRODUCTION

This paper summarizes some results obtained by applying the techniques of logic programming to problems in design automation, specifically in the design of CMOS transistor networks. We began this work for two main reasons: (1) to develop logic-programming idioms, evaluating the relative utility of various techniques, and (2) to demonstrate how logic programming can contribute effectively to design automation for digital circuits. We expect that many of the techniques discussed here apply more generally to other areas of design automation.

We have concentrated on high-level design issues rather than geometrical layout issues such as placement and routing. Experience suggests that while many months

Address correspondence to Mr. W. F. Clocksin, Computer Laboratory, University of Cambridge, Corn Exchange Street, Cambridge CB2 3QG, England.
Received 3 March 1986; accepted 6 June 1986.

THE JOURNAL OF LOGIC PROGRAMMING

52 Vanderbilt Ave., New York, NY 10017 0743-1066/87/$03.50

are required to solve design problems, layout is more of a mechanical and semiautomated task requiring far less time, on the order of weeks. Nevertheless, we have found routing a rich source of problems for investigating nonchronological backtracking techniques [18].

Also, we have concentrated on structural issues rather than behavioral issues such as verification [2]. Although we consider verification to be extremely important, there are two reasons why what we report here is an investigation of the extent to which reasoning can be done about circuits on the basis of circuit structure alone. First, this may have the effect of delaying the need to introduce behavioral specifications. Indeed, this certainly has the effect of automatically deriving some of the information usually specified manually as part of a behavioral specification, as we shall see below. Second, previous treatments of behavioral specifications have considered the special cases of no-delay and unit-delay [8] components. By contrast, a general delay model is able to represent the problems of sequential logic induced by the metastable nature of real circuits. I am currently working with a colleague on a general delay behavioral model on which we hope to report in due course. Using this model we intend to investigate the verification of behavioral specifications.

In what follows we shall first introduce the design methodology used. We use a hierarchical and modular design methodology for specifying the connection of abstract components of a system. Although this approach is a feature of many hardware description languages, our approach describes the system in terms of Horn formulae which are themselves amenable to formal manipulation. This is in contrast to programs written in most hardware description languages, which may be able to specify and simulate circuits, but which cannot be easily manipulated as objects themselves. Another aspect of the design methodology is given by the nondeterministic nature of the design task. There is more than one way to rewrite a given circuit, or to map it onto a group of components, or to lay it out as a pattern of wires and components. Design is an exploration of the space of possible choices, and we wish to reflect this in the way the programs operate. This usually implies the use of some sort of combinatorial search governed by a cost function.

Next we shall describe a number of problems we have solved using the methodology. We discuss circuit rewriting and gate assignment, which are good demonstrations of the design methodology, but which are of dubious practical value for electrical engineering reasons. Finally, we discuss the determination of signal flow and the specialization of standard designs, which have brighter prospects for application.

We shall use the Edinburgh PROLOG syntax for terms and formulae. This should be familiar to readers, but an introductory account is given in [3]. Although circuit specifications will be written using PROLOG syntax, the standard sequential right-to-left depth-first execution strategy is not assumed, and the specifications may be considered simply as Horn clauses. We shall also describe some programs, written in PROLOG, that manipulate these specifications. It will not prove necessary to discuss these programs in detail, but the standard execution strategy is assumed to be used for these.

2. CIRCUIT SPECIFICATION

A *circuit* is composed of a set of *modules* and connections between modules. With each module is associated a set of *ports* between which the connections are defined,

and which may be used for input and output. Modules can be composed hierarchically, in which modules are specified in terms of other modules. At the bottom of the hierarchy are primitive modules, the identity of which depends on the technology being used. For example, a VLSI designer might take transistors to be primitive; other digital designers might take logic gates or even individual packages to be primitive.

The first problem is to devise a representation of the circuit. We distinguish three methods, for which we have contrived some nomenclature: using terms to name functional circuits (the *functional* method), using a database of unit clauses to assert connection relations (the *extensional* method), and using formulae to represent hierarchical "plug-and-socket" circuit definitions (the *definitional* method). We shall describe each method in turn. The functional and extensional methods are shown to have sufficient disadvantages for our purposes that they are not considered further.

2.1. *The Functional Method*

The functional method can easily represent combinational acyclic circuits in which a single output signal is the *function* of several input signals. Modules are represented as ground compound terms, where a particular input port is associated with a particular argument of the term. Constant symbols are used to denote primitive modules which have no input (for example named input signals and power connections). The function symbol together with its arguments names the output of the module. The only connection relationship between modules is purely functional: the syntactic form of a given term determines the connections between modules. Thus, this technique does not make use of formulae. Consider the example in Figure 1(a), which shows a simple combinational circuit represented by the ground term

```
nor(nand(nor(not(b),a),c),nand3(a,b,not(c))).
```

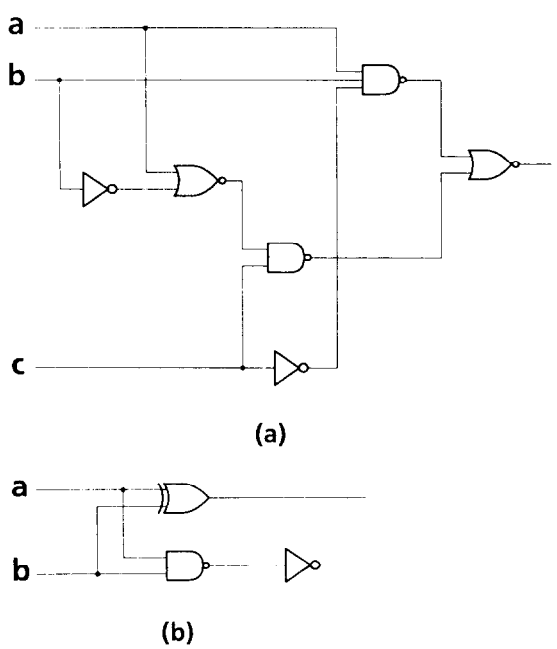


FIGURE 1.

In this example, input signals are named by the constant symbols `a`, `b`, and `c`. Modules are named by the 2-ary function symbols `nand` and `nor`, the 3-ary function symbol `nand3`, and the 1-ary function symbol `not`.

This representation permits processing of the circuit by using recursive descent to transform a given term into another, and it is used in a number of elementary treatments [3,17,21]. There are two main disadvantages to this technique. First, only acyclic circuits can be described by ground terms, and this precludes the specification of realistic sequential circuits. Second, a separate expression must be used to represent each output of a circuit. Figure 1(b) shows a circuit having outputs described by the pair of terms `xor(a,b)` and `not(nand(a,b))`. By introducing equations and metafunctional devices such as the μ-operator of Sheeran [19], it is possible to represent sequential circuits with cyclic connections and multiple outputs, whilst remaining within a functional-programming [9] context. Such a treatment brings this method closer to the definitional method described later; however, the usual use of the functional technique is restricted to ground terms, and does not include formulae. The two restrictions militate against using the functional technique for all but the most trivial of circuits found in practice.

2.2. The Extensional Method

The extensional method represents each module and connection as a unit clause in which constants are used to indicate the connections between modules. Figure 2 shows a circuit which can be specified by two predicates. The 3-ary predicate `module` describes the type of a module, a list of its input port names, and a list of its output port names. The binary predicate `connect` describes connections between ports, where the named port of a given module is represented as a ground 1-ary compound term consisting of a function symbol naming the module type together with an argument naming the port.

The circuit is thus described as the extension of the `module` and `connect` relations as follows:

```
module(xor,[a,b],[c]).
module(not,[a],[b]).
module(csrff,[s,c,r],[q])).

connect(xor(a),z).
connect(xor(b),x).
connect(xor(c),csrff(s)).
connect(xor(c),not(a)).
connect(not(b),csrff(r)).
connect(clock,csrff(c)).
connect(csrff(q),z)).
```

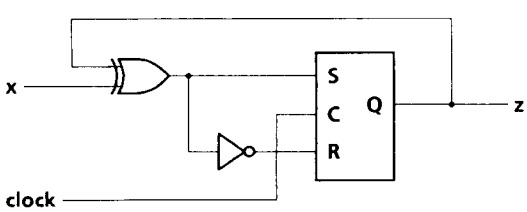


FIGURE 2.

Variations on this method are used in a number of systems [2, 10, 13]. It is usual to consider the `module` and `connect` relations as templates, and to augment their arguments with variables that stand for any instance of the module and connection. Additional arguments can be used to represent state variables, type information, and so forth.

The extensional method can accommodate arbitrary types of circuits such as multiple-output cyclic circuits. However, it has disadvantages stemming from the characteristic that modules are not represented by a single term. The modules and connections of a circuit are represented extensionally, with no syntactic relationship between them. The result is difficulty in efficiently carrying out certain operations such as circuit transformations, where in this case it is necessary to make awkward modifications to the database. Programs making use of this specification method often resort to explicit side effects—assertion and retraction of database clauses—in order to carry out simple transformations. By contrast, single terms as used by the other methods are easily rewritten without side effects, either by tree or by graph rewriting. Also, in the extensional method, there is less opportunity for modularity, as unit clauses contain no existentially quantified variables.

2.3. The Definitional Method

This method, which we shall treat in detail, represents a circuit as a first-order nonground formula. Modules having N ports are represented as n-ary predicate symbols. The modules in a given circuit are composed with a binary connective (here we use an infixed comma). Two or more ports within a given circuit that share a common connection are represented by coreferring (like-named) variables. It is convenient to write a module as a Horn clause in which the head of the clause represents the module to be defined, and the body of the clause is a composition of modules. The "`:-`" operator is reinterpreted here to mean "is defined by," and assumes the semantics of equivalence. The order of modules in the body of the clause is not important. The examples from above are shown as follows, where module names have been introduced:

```
combo(A,B,C,D) :-
            not(B,T1), nor(T1,A,T2), nand(C,T2,T3),
            not(C,T4), nand3(T4,A,B,T5), nor(T3,T5,D).

half_add(A,B,S,C) :- xor(A,B,S), nand(A,B,T1), not(T1,C).

seq_par(X,Clock,Z) :-
            xor(X,Z,T1), not(T1,T2), csrff(T1,Clock,T2,Z).
```

A specification of each primitive module is now required. For example, `not(`*a*`,`*b*`)` specifies an inverter, with input port *a* and output port *b*; `nand3(`*a*`,`*b*`,`*c*`,`*d*`)` specifies a three-input NAND gate for inputs *a*, *b*, *c*, and output *d*; `csrff(`*a*`,`*b*`,`*c*`,`*d*`)` specifies a clocked set-reset flip-flop with S-input *a*, clock *b*, R-input *c*, and output *d*. Specifications of these and other primitives will be listed shortly.

Variations of this method have been used previously [1, 6, 8, 15, 20]. The method has numerous advantages, especially when the circuits are represented as here by PROLOG clauses. First, the module name is explicitly part of the specification. This permits easy modular decomposition. For example, consider a three-bit subtractor,

which consists of a half subtractor and a pair of full subtractors:

```
half_sub(I1,I2,D,B0) :-
            xor(I1,I2,D), not(I1,T1), and (I2,T1,B0).

full_sub(A,B,BI,D,B0) :-
            xor(A,B,T1), xor(T1,BI,D), not(T1,T2),
            not(A,T3), nand(T2,BI,T4), nand(T3,B,T5),
            nand(T4,T5,B0).

three_sub(A0,A1,A2,B0,B1,B2,D0,D1,D2,T2) :-
            half_sub(A0,B0,D0,T0),
            full_sub(A1,B1,T0D1,T1),
            full_sub(A2,B2,T1,D2,T2).
```

Internal connections, which are named by variables not appearing in the head of the clause, are effectively "hidden." Such lexical scoping is a good engineering practice which is not provided by the extensional method described above. By inspection of the predicate-calculus equivalent formula, hidden variables are existentially quantified.

Further advantages of this specification technique are as follows:

Specifications can be directly executed by a PROLOG system.

Inherent representation of bidirectionality is obtained by the use of the logical variable. Bidirectionality is an important behavior of some components, such as pass transistors, and has not been explored in previous treatments. Input and output roles of ports can be constrained by extra clauses if necessary; however, the method shown here permits a representation of bidirectionality if required. Bidirectionality also influences signal-flow computations, discussed below.

An uninstantiated variable is a natural representation of the "floating" or high-impedance state.

Circuits can be specified by recursive composition.

We shall elaborate each of these points below.

3. DIRECT EXECUTION OF SPECIFICATIONS

Consider first the direct execution of the `half_sub` module. Definitions of the `xor`, `not`, and `and` primitives are given by the following unit clauses, which resemble the standard truth tables for these relations:

```
xor(0,0,0).
xor(0,1,1).
xor(1,0,1).
xor(1,1,0).

not(0,1).
not(1,0).
```

```
and(0,0,0).
and(0,1,0).
and(1,0,0).
and(1,1,1).
```

The constants 1 and 0 stand for logic high and logic low, respectively. Now the PROLOG goals for executing the half subtractor given above with all combinations of inputs to obtain differences D and borrow B are as follows, with the computer's output given in italics:

```
?- half_sub(0,0,D,B).
```
D = 0, B = 0

```
?- half_sub(0,1,D,B).
```
D = 1, B = 1

```
?- half_sub(1,0,D,B).
```
D = 1, B = 0

```
?- half_sub(1,1,D,B).
```
D = 1, B = 0

The four goals constitute a verification by exhaustive simulation. This is an unrealistic verification method in practice, and even generation of a smaller incomplete set of input test patterns can be unwieldy. We suggest that one alternative, still confined to simulation, is to use a method of "hypothetical states." Test patterns corresponding to inputs *and* outputs are queried, and the PROLOG system computes the possible conditions (by depth-first backtracking search of the circuit) under which the test pattern can be obtained. The test patterns may include uninstantiated variables, which correspond naturally to high impedance values. The clauses specifying a module are seen as constraints on the possible behavior of the module. For example, this query asks for the possible input and difference outputs for which the borrow output is 1. The only solution, computed by the PROLOG system, is given in italics:

```
?- half_sub(I,J,D,1).
```
I = 0, J = 1, D = 1

This "backwards" analysis of the circuit is possible because of the definition of circuits in terms of relations. Another example, which queries the conditions under which the first input is 1 and the second input is the same as the difference output, is as follows:

```
?- half_sub(1,J,J,B).
```
no

The PROLOG system correctly reports that no such state is possible for a correctly behaving `half_sub` module.

Next, consider the representation and simulation of logic functions using complementary transistors. We will make the standard simplifying assumption that the difference between gate-source capacitance and gate-drain capacitance is negligible. Where *G* stands for gate, *S* stands for source, and *D* stands for drain, we represent a *p*-type transistor as `ptrans(G,S,D)`, and an *n*-type transistor as

`ntrans(G,S,D)`. Definitions of the transistors, a complementary inverter, a complementary NAND gate, and a dynamic register cell are as follows:

```
ntrans(1,Y,Y).
ntrans(G,X,Y).

ptrans(0,Y,Y).
ptrans(G,X,Y).

invert(In,Out) :- ntrans(In,Out,0), ptrans(In,Out,1).

nand(I1,I2,Out) :-
            ptrans(I1,Out,1), ptrans(I2,Out,1),
            ntrans(I1,Out,W), ntrans(I2,W,0).

dreg(A,B,Load) :-
            ntrans(Load,A,X), invert(X,Y), invert(Y,B),
            ntrans(Z,B,X), invert(Load,Z).
```

If these specifications are directly executed as a PROLOG program, the logic gates behave correctly whether or not the inputs are instantiated. However, this transistor model also permits transistors to be driven "backwards": if the source and drain are equal, then a floating n-type gate will be forced high; a floating p-type gate will be forced low. Real transistors do not behave this way, and it is possible to modify the model accordingly. However, the simplicity of the model shown here is useful because it is not necessary to order the execution of goals when the clause is directly executed. Furthermore, the "backwards" behavior is needed to implement the method of hypothetical states. For example, the model will tell us the valid conditions under which the source and drain are equal. For an n-type transistor, there are two possibilities: (1) the gate is 1 (by the first clause of the definition of `ntrans`), or (2) the gate is floating (by the second clause) purely coincidentally. We believe that the nondeterministic nature of hypothetical reasoning about circuits is usefully captured by the logic programming method.

Our next example is a full adder constructed from complementary transistors, shown in Figure 3. The previous specifications of p- and n-transistors are used. An interesting feature of the adder is the use of two transistors which must conduct bidirectionally, depending on the input state. Direct execution of the `adder` module correctly simulates the bidirectional behavior. The modules are specified as follows:

```
adder(A,B,C,SUM,CARRY) :- carry_part(A,B,C,NCA,CARRY),
sum_part(A,B,C,NCA,SUM).

sum_part(A,B,C,NCA,SUM) :-
       ptrans(NCA,T1,1), ptrans(C,1,T5), ptrans(B,T1,T5),
       ptrans(A,T1,T2), ptrans(NCA,T5,T2), ptrans(T2,1,SUM),
       ntrans(A,T2,T3), ntrans(NCA,T2,T6), ntrans(T2,SUM,0),
       ntrans(B,T3,T6), ntrans(NCA,T3,0), ntrans(C,T6,0).

carry_part(A,B,C,NCA,CA) :-
       ptrans(A,T1,1), ptrans(B,T1,1), ptrans(A,T2,1),
       ptrans(C,T1,NCA), ptrans(B,T2,NCA), ptrans(NCA,1,CA),
       ntrans(C,NCA,T3), ntrans(B,NCA,T4), ntrans(NCA,CA,0),
       ntrans(A,T3,0), ntrans(B,T3,0), ntrans(A,T4,0).
```

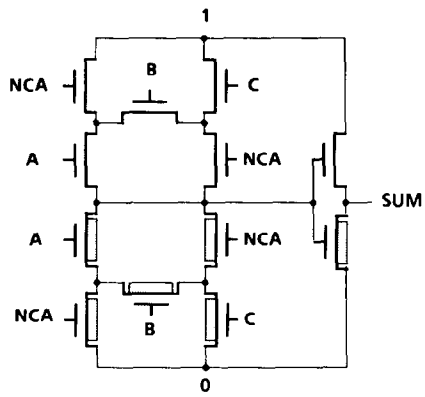
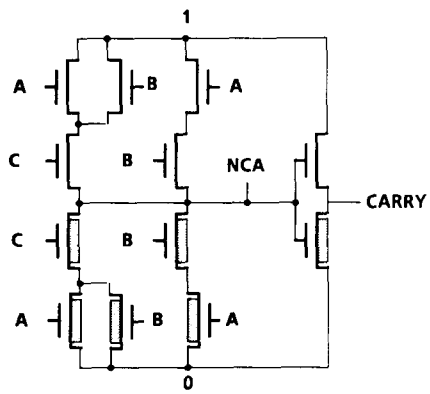


FIGURE 3.



It is easy to enumerate all 2^3 input states of this module, so this module can be verified by exhaustive simulation. Also, the method of hypothetical states can be used for "spot checking" of particular states of interest.

4. SEQUENTIAL SIMULATION BY DIRECT EXECUTION

We have seen how the definitional method can be used for direct execution of specifications. We now demonstrate how direct execution can be used for simulating sequential circuits. We begin by specifying the D-type flip-flop. The term `dff(`*D*`,`*C*`,`*Q*`,`*Q'*`)` represents the D-type flip-flop with input *D*, clock *C*, output *Q*, and next state *Q'*. We have left out the negated *Q* output available on some devices for convenience. The two unit clauses specifying `dff` are

```
dff(D,0,Q,Q).
dff(D,1,Q,D).
```

The first clause specifies behavior on a falling clock: the next state is the same as the current state. The second clause specifies behavior on a rising clock: the next state is the same as the *D* input.

The simplest sequential circuit, a divide-by-2 pulse divider, can be specified as follows (with the `not` module defined as above, not to be confused with the "not" predicate built into some PROLOG implementations):

```
div(C,Q,Z) :- not(Q,D), dff(D,C,Q,Z).
```

The literal `div(`C`,`Q`,`Q'`)` has a clock input C, a current state Q, and next state Q'. We can insert this module into a "test circuit" by writing a PROLOG procedure that recurs over an input list of clock pulses. The current state can be jammed with an initial state (say 0), and the output states are collected into an output list. The PROLOG goal `divide(P,I,Q)`, when given a pulse list `P` and initial state `I`, will construct an output pulse list `Q`. The definition of `divide` is

```
divide([],S[]).

divide([P|Ps],IS,[Q|Qs]) :- div(P,IS,Q), divide(Ps,Q,Qs).
```

Sample executions of this test circuit follow (computer's response in italics):

```
?- divide([1,1,1,1,1,1],0,Q).
```
Q = [1,0,1,0,1,0]

```
?- divide([0,1,0,0,1,1,0,0],0,Q).
```
Q = [0,1,1,1,0,1,1,1]

The next example is a sequential parity checker. On each clock pulse, the output provides an odd-parity check on however many data bits have been received by the serial input since the initial state of the circuit. The sequential parity checker is specified by the predicate `par(`C`,`D`,`Q`,`Q'`)` for clock input C, serial data input D, parity output Q, and next state Q', by the following definition (making use of `xor` as defined above):

```
par(C,D,Q,N) :- xor(D,Q,T), dff(T,C,Q,N).
```

We use the same technique of recurring over a list of clock pulses to form a test circuit `checker(`C`,`S`,`I`,`Q`)` for clock pulse list C, serial input list S, initial state I, and serial parity output list S:

```
checker([],S,I,[]).
checker([CI|Cs],[S|Ss],IS,[NS|L] :-
            par(C,S,IS,NS),
            checker(Cs,Ss,NS,L).
```

When the initial state is jammed to 0, an example goal together with the computer's reply is as follows:

```
?- checker([1,1,1,1,1,1],[1,0,0,1,1,0],0,Q).
```
Q = [1,1,1,0,1,1]

Note that, for the given input, odd parity is counted for the first three and last two clock pulses.

A final example is a three-bit synchronous Gray code counter depicted in Figure 4. Here the two combinational subcircuits are specified as separate clauses for procedures `neta` and `netb`. The `and` and `dff` modules are defined as above, and

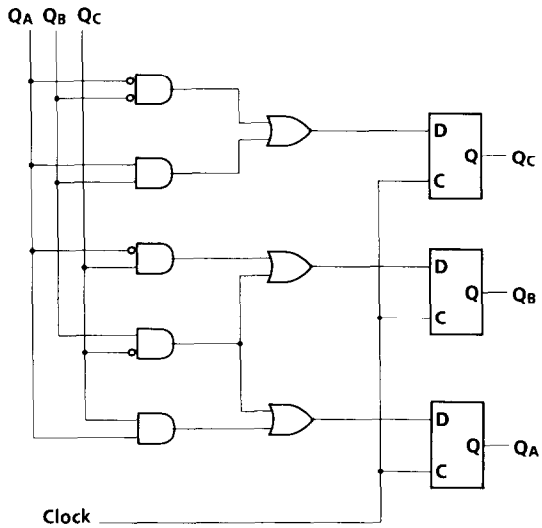


FIGURE 4.

the `or` module is easily defined in a similar way. A circuit state vector is represented by the compound term `s(`Q_a`,`Q_b`,`Q_c`)`, in which the state for each flip-flop is stored:

```
neta(A,B,C,Q1,Q2) :-
        and(A,C,T1), not(C,NC), and(B,NC,T2),
        not(A,NA), and(NA,C,T3), or(T1,T2,Q1),
        or(T2,T3,Q2).

netb(A,B,Q) :-
        and(A,B,T1), not(A,NA), not(B,NB),
        and(NA,NB,T2), or(T1,T2,Q).

gcc(C,s(Qa,Qb,Qc),s(Za,Zb,Zc)) :-
        neta(Qa,Qb,Qc,D1,D2), netb(Qa,Qb,D3),
        dff(C,D1,Qa,Za), dff(C,D2,Qb,Zb), dff(C,D3,Qc,Zc).
```

A test circuit is constructed as before:

```
testgcc([],S,[]).
testgcc([C|Cs],I,[N|Ns]) :- gcc(C,I,N), testgcc(Cs,N,Ns).
```

A query to test the circuit for nine pulses together with the result is

```
?- testgcc([1,1,1,1,1,1,1,1,1],s(0,0,0),Q).
```

Q = [s(0,0,1),s(0,1,1),s(0,1,0),s(1,1,0),s(1,1,1),s(1,0,1),s(1,0,0),s(0,0,0),s(0,0,1)]

It can be observed that the successive states represent an incrementing Gray code.

5. RECURSIVE DEFINITIONS

Recursion can be used to define parametric specifications of cascaded components. Consider the following example. The unit delay `ud` is a sequential component that

can be represented in the following way. The literal ud(I,S,S') is defined for input I, current state S, and next state S':

```
ud(0,0,0).
ud(1,0,1).
ud(0,1,0).
ud(1,1,1).
```

We may now connect N unit delays in series to produce an N-delay component. The component's state N-vector is represented as a list of length N. The literal nd(I,S,Q,S') is defined for input I, initial state N-vector S, output Q, and next state vector S':

```
nd(I,[],I,[]).
nd(I,[S|Ss],Q,[Z|Zs] :- ud(I,S,Z), nd(S,Ss,Q,Zz).
```

In this recursive definition, the output of each unit delay is passed to the input of the next, until the recursion terminates. This component may now be placed in a test circuit test(P,S,Q), where P is an input pulse list, S is an initial state N-vector, and Q is the output pulse list:

```
test([],S,[]).
test([P|Ps],S,[Q|Qs]) :- nd(P,S,Q,Z), test(Ps,Z,Qs).
```

A query to test the circuit delaying by three clock cycles a pulse list occupying eight clock cycles is

```
?- test([1,1,0,0,1,1,0,0],[0,0,0],Q).
```
Q = [0,0,0,1,1,0,0,1]

Now that we have covered the fundamentals of specifying circuits in PROLOG, we shall now turn to some of the problems that we have solved using this methodology.

6. GATE ASSIGNMENT

The purpose of gate assignment is to map a circuit onto a connected set of commercially available products, and then to estimate the cost of the resulting assembly. The output is a list of the products used (a parts list) and a list of connections to be made between the contact pins of the products (a net list). Although gate assignment is the lowest-level (closest to the hardware) problem discussed in this paper, we describe gate assignment first, since other problems discussed below depend on the gate-assignment program. At this point a technology choice must be made. In our programs we assume the use of 7400-series TTL dual-in-line packages wirewrapped on perforated board. The exact details of this choice are not important, and reflects only our familiarity with this technology. It would not be difficult to change the technology to, for example, standard cells as used in VLSI circuits.

To evaluate the cost of a given circuit, the cost function first maps the circuit onto a set of components, and then computes the cost of the components used plus a labor charge. The cost function accounts for the cost of each package used, the

area of board space occupied, and the cost of wiring and assembly. The cost function is highly discontinuous and nonlinear because several gates are packaged inside a given component. For example, the 7404 product contains six inverters; a type-7400 product contains four 2-input NAND gates. Depending on how many gates are allocated, allocation of another gate may cost nothing because a package of the right type has a spare gate, or may cost as much as a new package plus the area of board space it occupies.

Each primitive module (gate) in the circuit is allocated to a package using a nondeterministic "greedy" algorithm. The next available gate of the appropriate package is used, but if no gate is available, then an additional package must be added to the circuit. Pin connections are accumulated so that a net list can be produced when all gates have been assigned.

Figure 5(b) shows a circuit for a one-bit adder/subtracter **addsub**, a circuit which either adds or subtracts depending on the state of the **AS** input. Figure 5(a) depicts the definition of the half-adder **halfadd**, of which two instances are used in the definition of **addsub**. In the figure, each module is labelled with a unique module identifier of the form M*x*, which is used in a subsequent example and can be ignored here. Figure 6 shows the result of performing a gate assignment on this circuit. The program does not produce the pictorial output as shown, but produces a parts list and list of pin connections.

What makes this problem interesting is the nondeterministic assignment of gates to packages. If an attempt is made to backtrack a completed gate assignment, another assignment will be found. Because a greedy first-fit algorithm is used, the first solution will use a minimum number of packages. This is likely to be a low-cost solution, but this is not guaranteed. The last solution will allocate one gate per package, leaving many unused gates to produce a very high-cost solution. Intermediate solutions will choose another selection of gates, which may have an effect

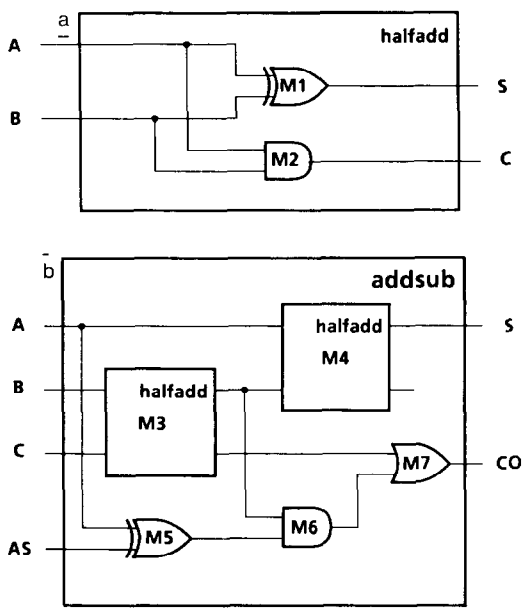


FIGURE 5.

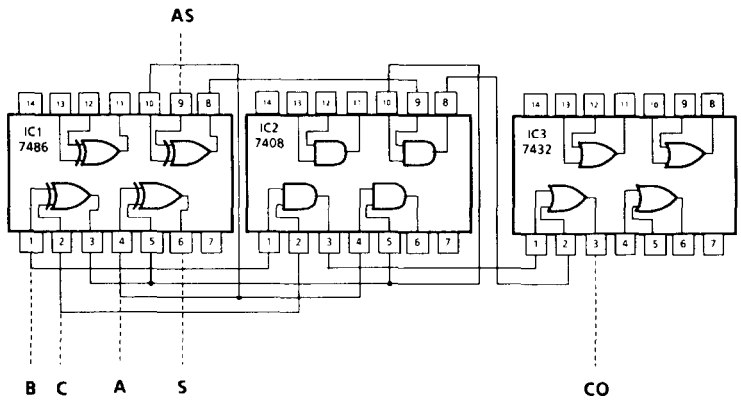


FIGURE 6.

on the ease with which connections can be made. For example, in the assignment in Figure 6, a result of the greedy algorithm is that gates are concentrated in the lower left-hand side of the diagram, leading to congested wiring. Another assignment (available on backtracking) would lead to a different selection of exclusive-or gates within the type-7486 package, which may produce less congested wiring. Behavior such as this has not been investigated further.

The allocation algorithm is implemented in only four PROLOG clauses, and the TTL database contains n clauses for each package type, where n is the number of gates in the package. The database currently contains clauses for seven popular package types, although more are easily added. Most of the clauses in the program are concerned with printing out the parts list and net list in a convenient format.

7. CIRCUIT REWRITING

The purpose of circuit rewriting is to construct a circuit which is logically equivalent to a given circuit. This technique is most often seen as Boolean simplification of circuits, but most rewriting techniques are of dubious validity because they do not preserve timing properties of the original circuit. Rewriting a circuit can easily introduce race hazards and change critical paths. This consideration is not always observed in elementary treatments [21], but techniques are available for rewriting circuits to be free of certain classes of timing hazards [22]. Nevertheless, rewriting is a demonstration of what can be done, and our treatment is intended to be tutorial at the risk of sacrificing timing constraints.

The input to our circuit rewriting program is a specification, which is treated as a graph to be rewritten. A database of transformation rules is used to rewrite subgraphs of the circuit. Two examples of transformation rules are shown in Fig 7. There are about two dozen such rules, covering most of the standard logic functions. The set of transformation rules is not intended to be complete in the formal sense, but typical of the common-sense rules known by circuit designers. Because there is considerable choice in the transformation rule and where it is to be applied, a cost function is used to evaluate the effectiveness of a rewrite. The goal is to rewrite the given circuit to another circuit of lower cost. The cost of a circuit is found simply by

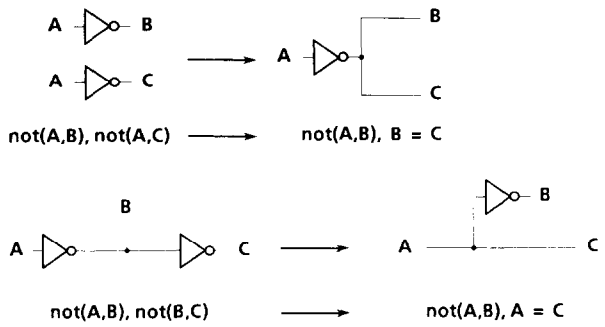


FIGURE 7.

performing a gate assignment (see Section 6), which estimates the cost required to build the circuit using a particular technology.

Because the cost function over the space of possible circuits and their rewrites is discontinuous and nonlinear with many low-cost states, we suspected that the usual methods for optimization by steepest descent would be unsuitable. We found many cases where it was necessary to introduce a locally expensive rewrite, knowing that only it would make possible the use of further rewrites that ultimately resulted in a low-cost circuit. Thus, using the terminology of Kirkpatrick [12], the program "anneals" the circuit rather than "quenches" it. Our annealing schedule is to alternate (phase-1) rewrites at any cost with (phase-2) rewrites at strictly decreasing cost. If a phase-2 rewrite is not possible given the transformation rules, then the system backtracks to try another phase-1 rewrite. The circuit is rewritten in this manner until no more rewrites can be performed according to the transformation rules.

Figure 8 shows a simple combinational circuit together with a graph depicting a cost reduction of $50p$ obtained by eight successive rewrites. The resulting circuit is essentially the minimal De Morgan equivalent of the circuit shown. The resulting circuit is of lower cost simply because only two types of component (inverters and 2-input NAND gates) are required, resulting in fewer packages. Notice that the second and fourth rewrites produced more expensive circuits (phase-1 rewrites).

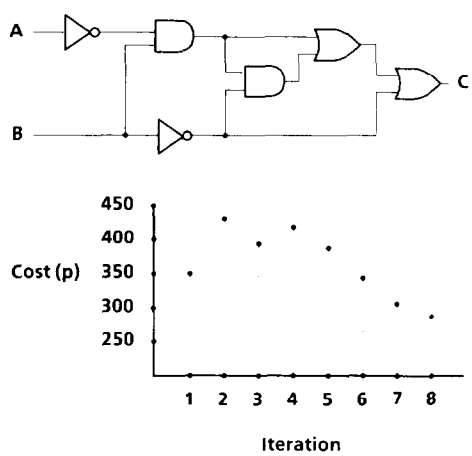


FIGURE 8.

After a *post mortem* manual investigation of the search tree it was found that the low-cost final solution could not have been obtained without performing the expensive rewrites.

8. SIGNAL FLOW ANALYSIS

Determining the signal flow through a network of transistors is often a necessary precursor before further circuit analysis can be carried out. For example, both Crystal [16] and TV [11] require flow analysis before they can proceed with timing analysis. Formal methods for circuit verification [8] and design simplification (see Section 9) depend on signal flow being specified in advance. Some design verification methods do not require a preliminary flow analysis, but proofs are easier to compute if the results of flow analysis are available [7]. One way to provide signal flow information is manually, by specifying the input and output roles of components' ports. Some systems [2, 8] rely on manual specifications entirely. However, automatic signal-flow determination can reduce the designer's workload, particularly for the case of MOS transistors, which, due to the symmetry of source and drain channels, are capable of conducting in different directions at different times.

The method described in this section is intended for hierarchically specified MOS transistor nets that contain bidirectional transistors, and where there is no information about any particular design methodology. A fuller account is given in [5]. Our method is data independent; it does not use knowledge of the values of inputs to a circuit. The analysis is static; it is only dependent on the circuit topology. In essence we are determining the flow of information through a net assuming all transistors are turned on. Although it is unlikely that this configuration will occur in practice, we are able to determine those transistors whose direction will always remain the same. Transistors which can support signal flow in either direction are labeled bidirectional. Clearly we label some transistors bidirectional which, due to the inputs provided, will only propagate signals in one direction in practice. However, the algorithm guarantees to correctly find all bidirectional transistors, and will not incorrectly label any unidirectional transistor. This procedure will be able to determine the direction of a large percentage of transistors in most circuits. It will not provide much useful information about a design that relies heavily on bidirectional transistors; however, such designs are unusual.

Signal flow may be viewed at a digital level as the propagation of logic levels through a network. Thus, power and ground connections are viewed as sources of signal flow. Similarly, inputs to a module and outputs from a module are sources and sinks of signal flow, respectively. Note the difference between signal flow and the more traditional view of current flow where power acts as a source of current flow and ground acts as a sink. For example, consider the CMOS inverter in Figure 9. The dotted arrows show the direction of current flow through the transistors; the solid arrows show the direction of signal flow. Using the definition of signal flow we can derive the input node and output node of the inverter, and this will be shown later.

The input to the direction finder is a circuit specification. We shall consider p- and n-transistors and power and ground supplies to be primitive. Ports may be sinks or sources of signal flow or they may be both. The task of determining the direction of signal flow can be viewed as a consistent-labeling problem [14]. To solve the

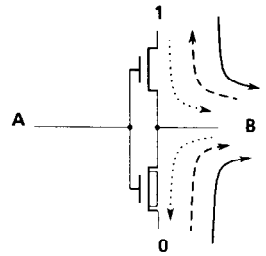


FIGURE 9. ···⟩: Current flow (by convention); ---⟩: electon flow; → signal flow.

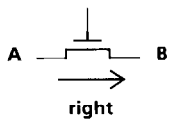
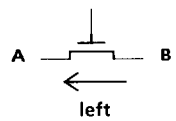


FIGURE 10.



problem, it is necessary to associate with each transistor a label drawn from the set {**right, left**}. Refer to Figure 10. If the two nongate terminals of a transistor are arbitrarily but consistently named *A* and *B*, the label **right** is assigned to a transistor for which terminal A is a sink of signal flow and terminal B is a source of signal flow. The label **left** is assigned to a transistor for which terminal A is a source of signal flow and terminal B is a sink of signal flow. For a circuit consisting of n transistors, the number of unconstrained labelings is 2^n. The problem is to assign labels to transistors so that a given constraint is satisfied. The constraint we use is that each node of a circuit must be connected to at least one sink and at least one source of signal flow. Moreover, if a node is directly connected to a power or ground, then at most one source of signal flow (the power or ground itself) is allowed. We call this constraint the *signal law*. The signal law is a consequence of Kirchhoff's current law. For a circuit containing bidirectional transistors, multiple solutions are admitted. That is, there may be more than one possible labeling of a circuit that satisfies the signal law. A bidirectional transistor will be labeled **right** in one solution and **left** in another.

It follows that the direction of port nodes (whether they are used as inputs or outputs) is an additional constraint on the solution. With our method there is a choice of whether to attempt to derive the direction of port nodes automatically (at the risk of obtaining a weak solution), or to use a specified direction of port nodes to constrain the search for labelings of transistors.

8.1. Implementation

The method uses depth-first search through the hierarchical circuit specification to exhaustively enumerate the directions of transistors. The method compounds the

direction finding of transistors with establishing the direction of nodes of nonprimitive modules. Unlike previous approaches, the whole circuit is *not* "flattened" into a set of primitive modules. First, the outermost module is decomposed into its constituent modules, then the direction of the ports of each of these modules is established, and finally it is confirmed that the hidden nodes of the circuit obey the signal law. If a module is not primitive, then the direction of its ports is established recursively.

The directions of ports of primitive modules, in this case *n*- and *p*-transistors, are determined as follows. Transistors have three ports: gate, source, and drain. A transistor's gate terminal is always a sink of signal flow. If a transistor specified as `ntrans(A,B,C)` is assigned the label **left**, then *B* is its source and *C* is its drain. Similarly, if it is assigned the label **right**, then *B* is its drain and *C* is its source. A transistor source is a source of signal flow; its drain is a sink. Either the label **left** or the label **right** is assigned to a transistor. This is a nondeterministic choice which may be reversed subsequently if it is found that the original choice leads to a violation of the signal law. Power and ground connections are always sources of signal flow.

The effect of this procedure is to try both directions of all transistors until the signal law is satisfied. Further backtracking will find all possible solutions. Transistors whose directions are the same in all possible solutions are unidirectional; the remaining transistors are labeled bidirectional. When the procedure succeeds, it is then possible to determine the capacitive loading of the port, and whether ports are used for input or output or are bidirectional.

We can further exploit the hierarchy of the circuit specification using the following technique. Once the directions of ports have been found for a type of submodule, the result can be stored in a library. This library would be consulted for each submodule when the procedure is invoked. Thus, signal flow only needs to be determined once for each type of submodule. In this context, trying both possible directions for all transistors in a circuit becomes reasonable for circuits with very large numbers of transistors. For example, a CMOS inverter is encountered in a circuit. Running our procedure on the inverter module once will set the direction of both transistors and establish the input and output. This result will then be stored, and each additional time an inverter is discovered, the input and output nodes are looked up in the library. Instead of four possible labelings for each inverter submodule, only one is considered. The benefits become greater as our library and the complexity of the submodules contained in that library grow.

8.2. Example

We will now show the result of applying this procedure to a simple example, the dynamic register `dreg` defined in Section 3. This example illustrates the utility of a library of submodules. As soon as the inverter module is encountered, its derived input and output specifications are stored in a library. Now only the direction of the two pass transistors need be considered. Thus only four possible label assignments are investigated.

From the method presented so far, it is impossible to ascertain a unique direction for the two pass transistors. There are three ways to deal with this problem. One is to leave the direction specified as bidirectional until the register is used in an

enclosing module. Input and output specifications derived for the enclosing module will constrain possible labeling for the pass transistors. Another way is to use specifications of the directions of ports. For example, in the `dreg` module, the specification that `A` is an input leads to the solution of the direction of the transistor directly connected to it. However, the specification that `B` is an output does not imply a single direction for the other transistor. Another approach is to incorporate a heuristic rule. This rule recognizes that two transistors whose gates are the complements of one another and that each have a channel connected to the same node have the same direction with respect to that node. This rule explicitly recognizes a 2-to-1 multiplex implemented with transistors, a configuration frequently found in designs. Such a rule is easily incorporated into our procedure.

Another example is the full-adder circuit of Figure 3. The module for computing the sum consists of twelve transistors, six of which, whose direction cannot be established, labeled as bidirectional. All of the transistors in the module for computing the carry are unidirectional. Our procedure also establishes that `A`, `B`, and `C` are always input ports and `Sum` and `Carry` are always output ports despite the different possible internal configurations.

8.3. Discussion

Signal-flow detection is able to recognize certain classes of specification errors. Circuit specifications that do not admit a labeling of transistor directions must be suspect as incorrectly specified. Such specifications contain at least one node which contains either all sinks or all sources of signal flow, an impossible situation in practice.

Techniques exploiting combinatorial enumeration are always suspected as being inefficient. We have already mentioned the use of the hierarchical specification of circuits to reduce the number of configurations to be examined. The number of different kinds of modules in the circuit becomes the important factor instead of the total number of transistors.

9. SPECIALIZATION

We next describe a technique, called specialization, for automatically removing a certain class of redundant modules in specifications. A fuller account of this technique appears in [4]. Modules are subject to specialization when they provide outputs that are not used in a circuit. This situation is not uncommon in, for example, "standard cell" designs available from a number of commercial sources. As with the previous method for determining signal flow, new specialized definitions of cells are collected in a library as they are constructed, so that detailed processing need not be repeated when the same specialization is required elsewhere in a circuit.

The use of standard cells has increased the convenience of designing VLSI circuits. Standard cells are an example of the modular design methodology used in this paper. However, circuits constructed from standard cells generally contain more primitive components (transistors) than purpose-built circuitry because the standard-cell library can only provide a manageably small number of general-purpose cells. Suppliers of standard-cell libraries cannot predict in advance the precise

functionality required by the designer. A consequence is that some of the functionality of general-purpose cells is not required in a design, causing "overdesigned" circuits to be produced.

The abstraction boundaries provided by the standard-cell approach exist for the convenience of the designer. However, once the circuit is out of the designer's control, it is desirable to automatically redesign the circuit to more easily meet constraints imposed by technology. Such redesign would involve removing redundant circuitry that exists for the two reasons given above, and may result in redefining abstraction boundaries within a circuit. Specialization removes redundant circuitry by automatically generating more specialized circuits when given a circuit defined in terms of general-purpose modules. This technique can be seen as a compromise between the ultimate goal of silicon compilation and the need to use existing methodologies similar to standard cells.

Specialization is not the same as rewriting a circuit to perform, for example, Boolean minimization or strength reduction. Specialization will not remove parts along a datapath having an output that is used elsewhere in a circuit. The reason for this more conservative approach is to prevent the violation of designed-in timing constraints owing to propagation delay through critical paths. Instead, specialization deals with overgeneral design which does not contribute to the final outputs of the circuit.

The program works by selectively ignoring abstraction boundaries imposed by the module definitions and by the hierarchical specification of the circuit, automatically designing new module definitions where required. We observe that much redundant circuitry can be identified by tracing unused outputs back through the circuit. Although this tracing is a simple operation that produces a specialized circuit from one constructed from general-purpose cells, this operation when performed manually is often tedious and error prone as the designer can become lost by the complexity of large circuits.

This technique can be computationally complex, but in our implementation, efficiency is achieved by exploiting the hierarchical nature of the circuit specification to suppress unnecessary detail, in the same way as for the signal-flow problem (Section 8). The computations required to specialize a given type of cell in a given way are performed only once. The new definition is stored in a library to be used again when required. Thus, the work done in specializing a complete circuit is linear in the number of types of cell in its design, rather than linear in the number of components (transistors).

A module can be specialized if it contains at least one unused output. The trivial case of specialization is when the unused output is the only output of the module. In this case, the entire module can be removed from the circuit with impunity. Removal of the module may result in the disconnection of outputs from other modules, which can be specialized in turn. Specialization thus becomes a tree search rooted in each unused output of a module, and propagating back through the inputs to the module. The tree search propagates though the circuit, inspecting only those parts of the circuit on a direct path to the unused output.

The search through the circuit must also take account of hierarchical structure. Consider first primitive modules, where specialization results in no change in the hierarchy. For primitive modules having all outputs unused, the trivial case holds, and the module can be removed. For primitive modules having at least one used

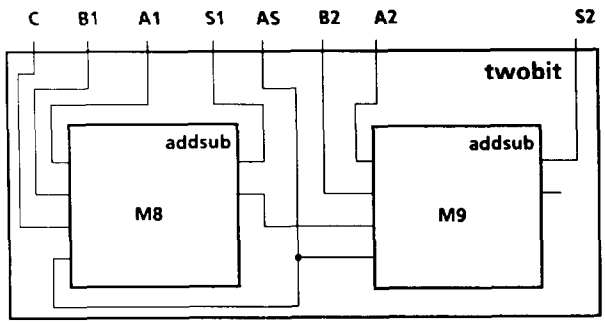


FIGURE 11.

output, no change can be made, and the search terminates. Now consider nonprimitives. If all outputs of a nonprimitive module are unused, then the trivial case again holds, and the module is removed. However, given a nonprimitive module in which some but not all outputs are unused, it may be necessary to recursively descend the hierarchy into the module definition to determine whether any submodules sourcing the unused outputs can be removed from the definition of the module. Removing submodules may have the effect of rendering redundant some of the inputs to the module, and the search proceeds from there after emerging from the recursion. But, before the search continues, it is necessary to redefine the modified module as a specialized version of the original module. In our implemented system, redefined modules are placed in a "library" from which they can be accessed in case another module of the same specialization is found elsewhere in the circuit. If an attempt is made to specialize a module having a given pattern of unused outputs, the library is first scanned for a definition, which, if found, immediately replaces the module under consideration.

This can be explained by means of an example shown in Figure 11, taken from [4]. Consider the two-bit adder/subtractor `twobit` which is defined using the modules shown in Figure 5. The unique module identifiers of the form Mx are for explanatory purposes only. The carry output of the last stage of `twobit` is unused.

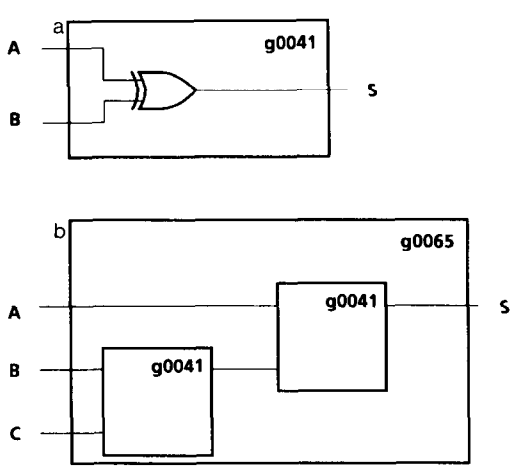


FIGURE 12.

Given the top-level goal of specializing `twobit`, the program proceeds as follows:

1. Inspect the modules of `twobit`, finding an `addsub` (M9) with an unused output. Recursively enter the definition of `addsub` M9 with the goal of specializing it for this particular instance.
2. The following modules are removed from `addsub` M9: the `or` gate M7, the `and` gate M6, and the `xor` gate M5. The half adder `halfadd` M3 inside `addsub` M9 now has a disconnected carry output, so recursively enter the definition of `halfadd` M3 with the goal of specializing it for this particular instance.
3. The `and` gate M2 is removed from `halfadd` M3. No other modules with unused outputs remain in `halfadd` M3, so now preparations are made to return to the next higher level of the circuit hierarchy. The specialized half adder is added to the library with a generated unique name, in this case `G0041` [see Figure 12(a)]. The `halfadd` M3 module is now replaced by module `G0041`.
4. The one remaining module in `addsub` M9 having an unused output is another `halfadd`, M4. Because a half adder with an unused carry output exists in the library as module `G0041`, a copy of `G0041` replaces the `halfadd` M4. No other modules with unused outputs remain in `addsub` M9, so now preparations are made to return to the next higher level of the circuit hierarchy. The specialized `addsub` is added to the library with a generated unique name, in this case `G0065` [see Figure 12(b)]. The module `addsub` M9 in `twobit` is now replaced by module `G0065`.
5. Propagation continues along the `AS` input of `twobit`, but search terminates because the `AS` line sources a used input (`AS` of `addsub` M8). No other modules with unused outputs remain in `twobit`, so now control returns to the top level of the circuit hierarchy.

By convention, the top-level circuit is not added to the library, so the final result is the circuit shown in Figure 13. To observe what specialization has occurred, compare Figure 5(a) and (b) with Figure 12(a) and (b), respectively.

The specialization program has been tested on a variety of small modules, and these are discussed in [4] together with electrical-engineering implications of the method. For each module in a circuit, the specializer needs to know the direction (whether input or output) of each port, but this information is given by the

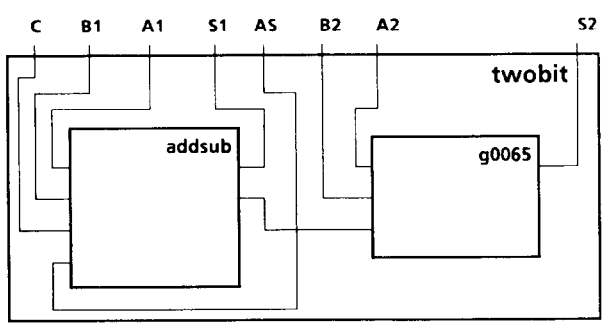


FIGURE 13.

signal-flow technique of Section 8. The specialization program can cope with bidirectional circuit modules, such as the bidirectional transistors in the adder of Figure 3. The specializer is able to make specializations of this circuit, but the bidirectional transistors are never removed. For example, if the **SUM** output were unused, then the only action available would be to remove the two transistors making up the inverter at the last stage.

10. CONCLUSIONS

After much experimentation, the definitional method of specification has emerged as being more convenient than either the functional method or the extensional method. In particular, specifications can be directly executed as PROLOG programs; the logical variable plays a key role in implementing bidirectionality and high-impedance states; and circuits can be specified recursively.

Given the PROLOG context of combinatorial enumeration of choices, it is important to devise methods whose complexity does not depend exponentially on the number of primitive components in the system. By using the concept of libraries we have reduced the complexity to depending on the number of different modules used in a system. Constraints on choice are also propagated throughout the circuit with the result of reducing computation, as seen in determining signal flow, but the detailed characteristics of this interaction are at the moment mysterious.

We can identify some areas deserving further treatment. First of all, we have ignored the timing characteristics of circuits. This is most obvious in circuit rewriting, where timing properties of the circuit are not guaranteed to be preserved. Previous delay models are of limited value; hence our desire to investigate a general delay model. We have not yet considered algebraic verification of behavioral specifications, for reasons described in Section 1. Yet, the methods described in this paper are of use in connection with the verification task.

Apart from the general delay model, our current investigations are focused on applying the methods outlined in this paper to the specification of mechanical parts and the verification of process control.

I thank Don Gaubatz, Mike Gordon, Miriam Leeser, Ben Moszkowski, and others for much discussion. The CMOS adder is due to Inder Dhingra. Miriam Leeser implemented two versions of the signal-flow determination program.

REFERENCES

1. Batten, J. W., Prolog: Its potential for hardware description and verification, Dept. of Computation, UMIST, 1983.
2. Barrow, H. G., VERIFY: A Program for Proving Correctness of Digital Hardware Designs, *Artificial Intelligence* 24:437–491 (1984).
3. Clocksin, W. F. and Mellish, C. S., *Programming in Prolog*, Springer, 1981.
4. Clocksin, W. F., Automatic Specialization of Standard Designs, *Comput. J.*, to appear.
5. Clocksin, W. F. and Leeser, M. E., Automatic Determination of Signal Flow through MOS Transistor Networks, *Integration* 4:53–63 (1986).
6. Fujita, M., Logic Design Assistance with Temporal Logic, Draft Thesis, Univ. of Tokyo, 1983.

7. Gordon, M. J. C., A Very Simple Model of Sequential Behaviour of nMOS, in: J. P. Gray (ed.), *VLSI* '81, Academic, 1981.
8. Gordon, M. J. C., LCF-LSM: A System for Specifying and Verifying Hardware, Technical Rep. 41, Computer Lab., Univ. of Cambridge, 1983.
9. Henderson, P., *Functional Programming*, Prentice-Hall, 1980.
10. Horstmann, P. W., Expert Systems and Logic Programming for CAD, *VLSI Design*, Nov. 1983.
11. Jouppi, N. P., TV: An nMOS Timing Analyzer, in: *Proceedings of the 3rd Caltech VLSI Conference*, 1983, pp. 71–76.
12. Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P., Optimization by Simulated Annealing, *Science* 220:671–680 (1983).
13. Kollaritsch, P. W. and Weste, N. H. E., A Rule-Based Symbolic Layout Expert, *VLSI Design*, Aug. 1984, pp. 62–66.
14. Mackworth, A. K., Consistency in Networks of Relations, *Artificial Intelligence* 8:99–118 (1977).
15. Moszkowski, B. C., A Temporal Logic for Multilevel Reasoning about Hardware, *Computer* 18(2):10–19 (1985).
16. Ousterhout, J. K., Crystal: A Timing Analyzer for nMOS VLSI Circuits, in: *Proceedings of the 3rd Caltech VLSI Conference*, 1983, pp. 57–69.
17. Sammut, R. A., and Sammut, C. A., Prolog: A Tutorial Introduction, *Austral. Comput. J.* 15(2):42–51 (1983).
18. Shanahan, M. P., Intelligent Backtracking and Routing, Computer Lab., Univ. of Cambridge, 1986.
19. Sheeran, M. μFP, An Algebraic VLSI Design Language, D. Phil. Thesis, Univ. of Oxford, 1984.
20. Svanajes, D. and Aas, E. J., Test generation through logic programming, *Integration* 2:49–67 (1984).
21. Wos, L., Overbeek, R., Lusk, E., and Boyle, J. *Automated Reasoning*, Prentice-Hall, 1984.
22. Zissos, D., *Logic Design Algorithms*, Oxford U.P., 1972.